Implementing OCaml APIs in Coq

Joshua M. Cohen
jmc16@princeton.edu
Princeton University

Abstract

Extraction is a good way to produce verified programs; the extracted code can be linked with hand-written OCaml to produce an executable. But this is not sufficient to implement (stateful, exception-throwing) OCaml APIs, whose types and behavior must be preserved exactly for client compatibility. We propose a lightweight design pattern for implementing such functions by carefully modifying extraction to provide support for features such as exceptions and mutable state. The resulting programs are executable both within Coq and in OCaml, exactly matching the expected OCaml interface. We provide a small library to enable programming with this pattern and demonstrate on 3 examples: a subset of OCaml's `List` library, a mutable counter, and a stateful term API that generates unique variable names for α-conversion and safe substitution.

1 Introduction

Coq's extraction mechanism, which translates Gallina to OCaml or Haskell, has been crucial in developing usable verified programs including CompCert [5], the FSCQ file system [4], and the CertiCoq compiler [2]. Such programs typically link the extracted code with additional hand-written code to enable interaction with the outside world (e.g. a command line interface, or file I/O).

However, most real-world programs are built atop APIs of existing libraries, and often provide their own APIs for clients. Even many executable OCaml programs follow this approach, including formal methods tools like Coq itself,[1] the Why3 deductive verification framework,[2] the Frama-C C verifiation toolchain,[3] and the Alt-Ergo SMT solver.[4] These tools are built atop more basic standard libraries (e.g. Stdlib, Batteries, and Jane Street's Core). Being able to implement such APIs in Coq enables *incremental verification* – one could replace some API functions with proved-correct versions and the client would get the benefit of verification for free.

But such APIs cannot in general be implemented in Coq. For example, the hd function in OCaml's `List` library throws an exception if the list is empty:

```
val hd : 'a list -> 'a
let hd = function
[] → failwith "hd"
| a::_ → a
```

[1]https://coq.inria.fr/doc/master/api/
[2]https://www.why3.org/api/
[3]https://frama-c.com/api/
[4]https://ocamlpro.github.io/alt-ergo/latest/API/index.html

Clients may rely on this behavior, but it is impossible to write an axiom-free function with this type in Coq. Exceptions are not the only incompatibility; other OCaml features that are difficult or impossible to write in Coq include mutable state, efficient machine-length integers, I/O, nontermination, and private types. Here, we focus on exceptions, state, and ints, proving an axiom-free library of functions to enable such implementations in Coq. The main idea is to *modify the extraction* to represent features differently in Coq and OCaml; this increases the trusted code base, but results in computable and idiomatic code on both sides.

2 A Library for Implementing APIs

2.1 Error Handling

We demonstrate our approach for exceptions, aiming to implement the above hd example. In Coq, we represent exceptions using the error monad from the coq-ext-lib library. First, we define a type of exceptions; like in OCaml, we can create new instances on the fly:

```
Record errtype : Type := { errname : string;
     errargs: Type; errdata : errargs}.
Definition mk_errtype name {A} (x: A) :=
  {|errname := name; errargs := A;
    errdata := x|}.
Definition Failure (msg: string) : errtype
    := mk_errtype "Failure" msg.
```

The error monad is just a sum type of either the result or the error; we use the monad functions (return, bind, etc) from coq-ext-lib:

```
Definition errorM A : Type :=
  Datatypes.sum errtype A.
Definition err_ret {A} (x: A) : errorM A :=
  ret x.
Definition err_bnd {A B} (f: A→ errorM B)
  (x: errorM A) : errorM B := bind x f.
Definition throw : ∀ {A} (e: errtype),
  errorM A := fun A e ⇒ raise e.
```

When extracting, errtype becomes exn, errorM A becomes just A, and monad functions are extracted accordingly (ret becomes the identity, bind becomes function application, and throw becomes raise):

```
Extract Constant errorM "'a" ⇒ "'a".
Extract Inductive errtype ⇒ exn [""].
Extract Inlined Constant err_ret ⇒
  "(fun x → x)".
Extract Inlined Constant err_bnd ⇒ "(@@)".
Extract Inlined Constant throw ⇒ "raise".
Extract Inlined Constant Failure ⇒ "Failure"
    .
```

Then we can write a function in Coq in this error monad. As long as we use the above functions (err_ret, throw, etc), the resulting OCaml code is exactly what we expect:

```
Definition hd {A: Type} (l: list A) :
  errorM A := (*Coq*)
match l with
| nil ⇒ throw (Failure "hd")
| x :: _ ⇒ err_ret x
end.
let hd = function (*OCaml*)
| [] → raise (Failure "hd")
| x::_ →  x
```

Furthermore, we can reason about hd in Coq as usual and compute with it via Coq's compute mechanisms.

2.2 Integers and Mutable State

To implement efficient integers and mutable state, we follow the same basic approach: providing different implementations in Coq and OCaml and carefully relating them during extraction. We provide two flavors of integers: 31-bit integers, which we implement in Coq using CompCert's Integers library and in OCaml by int, and unbounded integers, which we implement in Coq as Z and in OCaml using the efficient Zarith library. The latter is easier to work with in proofs, but many existing OCaml APIs (e.g. List.length) use the former.

For mutable state, we use coq-ext-lib's state monad in Coq, which we encapsulate in a generic State module. In OCaml, we implement this with a mutable reference. However, the behavior differs: in Coq we can run a state from *any* initial state, but with a mutable reference, the initial state is fixed. Naive translation is unsound: for example, given function incr that increments a counter and returns the new value, runState incr 0 = runState incr 0 in Coq but not in OCaml. To avoid this, we parameterize the State module by an initial value and only permit the user to run the state starting from this value; whenever runState is called, the state is also reset to this initial value in both Coq and OCaml. This models the behavior of Coq: each time we force the state to evaluate, the history is cleared and the state resets.

3 Examples

We first demonstrate our methods by implementing both a mutable counter and several OCaml List library functions using exceptions and integers. In each case, we program against an idiomatic .mli interface. We prove several theorems about the List functions and write small Coq and OCaml client programs calling each API.

As a more interesting case study, we implement a task particularly difficult to implement efficiently in a pure language: generating globally unique names. We give a small API for terms with integer-valued variables based on the design of the Why3 [3] term API[5] and define a semantics under a given variable context. The (simplified) API is:

```
type var
type tm_bound
type tm = private ... | Tvar of var
  | Tlet of tm * tm_bound
val create_var : string → var
val t_open_bound: tm_bound → (var * tm)
val t_close_bound: var → tm → tm_bound
val sub_t : var → tm → tm → tm
```

Our API implements variable binding as Why3 does: bindings and variables are abstract to clients, and the only ways to create variables are via create_var and t_open_bound, which use a global counter to assign each newly created variable a fresh tag. When substituting, t_open_bound creates fresh bindings to avoid capture.[6] We implement this in Coq via our State module; the type of create_var is state Z var. To reason about our substitution function, we implement a version of the Hoare State Monad [8], equipping our state with pre- and post-conditions. This allows us to prove the correctness of substitution against our semantics.

Finally, we again give client programs: in OCaml, we write a function to eliminate let bindings, while in Coq we prove that a single let-elimination is correct; in both cases we demonstrate on concrete examples.

4 Limitations and Related Work

We take a pragmatic, lightweight approach to combining monads with imperative programs. In contrast, Ynot [6] provides a heavy-duty framework for imperative programming in Coq, axiomatizing the stateful operations. Similarly, Abrahamsson et. al. produce proved-correct imperative CakeML programs from monadic HOL code [1].

Our design pattern is simple and lightweight; this introduces several limitations:

- We must trust the modified extraction directives (though we do not add any axioms to Coq's logic).
- It is difficult to control opacity in Coq; we cannot ensure that user calls only functions from the monadic interface.[7] The situation would be improved if Coq had a .mli-like-construct [7].
- The resulting OCaml code satisfies the desired API but often has incorrect imports, so we post-process the resulting files with dune.

Nevertheless, this method is effective for producing programs that can be executed in Coq and OCaml, used by clients in both Coq and OCaml, and reasoned about in Coq. Our code is available at github.com/joscoh/coq-ocaml-api.

[5]https://www.why3.org/api/Term.html

[6]To prove the termination of functions recursing on t_open_bound, we need a dependently typed version of bind for state that remembers the argument the continuation is applied to.

[7]For example, a user could execute the underlying state monad, violating our runState properties. To warn against this, we name the internal function st_run_UNSAFE.

References

[1] Abrahamsson, O., Ho, S., Kanabar, H., Kumar, R., Myreen, M. O., Norrish, M., and Tan, Y. K. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *Journal of Automated Reasoning 64*, 7 (Oct. 2020), 1287–1306.

[2] Anand, A., Appel, A. W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O. S., Sozeau, M., and Weaver, M. CertiCoq: A Verified Compiler for Coq. In *CoqPL'17: The Third International Workshop on Coq for Programming Languages* (Paris, France, Jan. 2017).

[3] Bobot, F., Filliâtre, J.-C., Marché, C., and Paskevich, A. Why3: Shepherd Your Herd of Provers. p. 53.

[4] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, Oct. 2015), SOSP '15, Association for Computing Machinery, pp. 18–37.

[5] Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (July 2009), 107–115.

[6] Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., and Birkedal, L. Ynot: dependent types for imperative programs. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, Sept. 2008), ICFP '08, Association for Computing Machinery, pp. 229–240.

[7] Swasey, D., Giarrusso, P. G., and Malecha, G. A Case for Lightweight Interfaces in Coq. In *CoqPL'22: The Eighth International Workshop on Coq for Programming Languages* (Philadelphia, US, Jan. 2022).

[8] Swierstra, W. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics* (Berlin, Heidelberg, 2009), S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., Springer, pp. 440–451.